CS103
Winter 2019

Cynthia Lee
Keith Schwarz

Problem Set 7

What can you do with regular expressions? What are the limits of regular languages? In this problem set, you'll explore the answers to these questions along with their practical consequences.

As always, please feel free to drop by office hours, ask on Piazza, or send us emails if you have any questions. We'd be happy to help out.

Good luck, and have fun!

Due Friday, March 1st at 2:30PM

Problem One: Designing Regular Expressions

Below are a list of alphabets and languages over those alphabets. For each language, write a regular expression for that language. ***Please use our online tool to design, test, and submit your regular expressions. Typed or handwritten solutions will not be accepted.*** To use it, visit the CS103 website and click the "Regex Editor" link under the "Resources" header. If you submit in a pair, please tell us in your GradeScope submission who submitted your answers to this question. Also, as a reminder, please test your submissions thoroughly, since we'll be grading them with an autograder.

i. Let $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}\}$. Write a regular expression for the language $\{ w \in \Sigma^* \mid$ the letters in w are sorted alphabetically $\}$.

ii. Write a regular expression for the complement of the language from part (i) of this problem.

As with NFAs, there's no simple way to start with a regex for a language L and to turn it into a regex for $\overline{L}$.

iii. On Unix-style operating systems like macOS or Linux, files are organized into directories. You can reference a file by giving a *path* to the file, a series of directory names separated by slashes. For example, the path `/home/username/` might represent a user's home directory, and a path like `/home/username/Documents/PS7.tex` might represent that person's solution to this problem set. Paths that start with a slash character are called *absolute paths* and say exactly where the file is on disk. Paths that don't start with a slash are called *relative paths* and say where, relative to the current folder, a file can be found. For example, if I'm logged into my computer and am in my home folder, I could look up the file `Documents/PS7.tex` to find my solution to this problem set.

The general pattern here is that a file path consists of a series of directory or file names separated by slashes. That path might optionally start with a slash, but isn't required to, and it might optionally end with a slash, but isn't required to. However, you can't have two consecutive slashes.*

Let $\Sigma = \{\texttt{a}, \texttt{/}\}$. Write a regular expression for $L = \{ w \in \Sigma^* \mid w$ represents the name of a file path on a Unix-style system $\}$. For example, $\texttt{/aaa/a/aa} \in L$, $\texttt{/} \in L$, $\texttt{a} \in L$, $\texttt{/a/a/} \in L$, and $\texttt{aaa/} \in L$, but $\texttt{//a//} \notin L$, $\texttt{a//a} \notin L$, and $\varepsilon \notin L$.

Fun fact: this problem comes from former TA Amy Liu, who fixed a bug in industrial code that arose when someone wrote the wrong regex for this language. Oops.

iv. Suppose you are taking a walk with your dog on a leash of length two. Let $\Sigma = \{\texttt{y}, \texttt{d}\}$ and let $L = \{ w \in \Sigma^* \mid w$ represents a walk with your dog on a leash where you and your dog both end up at the same location $\}$. For example, we have $\texttt{yyddddyy} \in L$ because you and your dog are never more than two steps apart and both of you end up four steps ahead of where you started; similarly, $\texttt{ddydyy} \in L$. However, $\texttt{yyyyddd} \notin L$, since halfway through your walk you're three steps ahead of your dog; $\texttt{ddyd} \notin L$, because your dog ends up two steps ahead of you; and $\texttt{ddyddyyy} \notin L$, because at one point your dog is three steps ahead of you. Write a regular expression for L.

*Note that, unlike Problem Set Six, you and your dog **must** end at the same position.*

v. Let $\Sigma = \{\texttt{M}, \texttt{D}, \texttt{C}, \texttt{L}, \texttt{X}, \texttt{V}, \texttt{I}\}$ and let $L = \{ w \in \Sigma^* \mid$ w is number less than 2,000 represented in Roman numerals $\}$. For example, $\texttt{CMXCIX} \in L$, since it represents the number 999, as are the strings

* In some cases you technically *can* have multiple consecutive slashes, but we'll ignore that for now.

`L` (50), `VIII` (8), `DCLXVI` (666), `CXXXVII` (137), `CDXII` (412), and `MDCXVIII` (1,618). However, we have that `VIIII` $\notin L$ (you'll never have four `I`'s in a row; use `IX` or `IV` instead), that `MM` $\notin L$ (it's a Roman numeral, but it's for 2,000, which is too large), that `VX` $\notin L$ (this isn't a valid Roman numeral), and that `IM` $\notin L$ (the notation of using a smaller digit to subtract from a larger one only lets you use `I` to prefix `V` and `X`, or `X` to prefix `L` and `C`, or `C` to prefix `D` and `M`). The Romans didn't have a way of expressing 0, so to make your life easier we'll say that $\varepsilon \in L$ and that the empty string represents 0. (Oh, those silly Romans.) Write a regular expression for L.

(As a note, we're using the "standard form" of Roman numerals. You can see a sample of numbers written out this way via this link.)

Problem Two: Finite Languages

A language L is called ***finite*** if L contains finitely many strings (that is, $|L|$ is a natural number).

i. Given a finite language L, explain how to write a regular expression for L. Briefly justify your answer; no formal proof is necessary. This shows that all finite languages are regular.

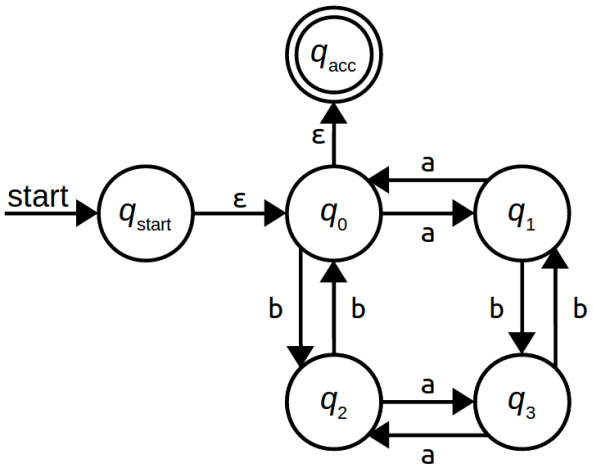
Watch for edge cases!

ii. Look up the ***trie*** data structure on Wikipedia. Explain, in your own words, how a trie can be thought of as an NFA for a finite language. This gives another view on why finite languages are regular.

Problem Three: State Elimination

The state elimination algorithm gives a way to transform a finite automaton (a DFA or NFA) into a regular expression. It's a really beautiful algorithm once you get the hang of it, so we thought that we'd let you try it out on a particular example.

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and let $L = \{ w \in \Sigma^* \mid w$ has an even number of `a`'s and an even number of `b`'s$\}$. Below is a finite automaton for L that we've prepared for the state elimination algorithm by adding in a new start state q_{start} and a new accept state q_{acc}:



We'd like you to use the state elimination algorithm to produce a regular expression for L.

i. Run two steps of the state elimination algorithm on the above automaton. Specifically, first remove state q_1, then remove state q_2. Show your result at this point.

Go slowly. Remember that to eliminate a state q, you should identify all pairs of states q_{in} and q_{out} where there's a transition from q_{in} to q and from q to q_{out}, then add shortcut edges from q_{in} to q_{out} to bypass state q. Remember that q_{in} and q_{out} can be the same state. If you've done everything right at the end of this stage, none of the transitions you have at this point should have Kleene stars in them.

ii. Finish the state elimination algorithm, showing your work. ***Submit your resulting regular expression through our regular expression tool*** along the lines of what you did in Problem One.

You should start seeing Kleene stars appearing as you remove the remaining states.

Not sure whether you have the right answer? ***Test your result thoroughly!***

iii. Without making reference to the original automaton given above, give an intuitive explanation for how the regular expression you found in part (ii) works.

The regex you've found works by matching strings in a very creative way. Tell us what that way is.

Problem Four: At Least Three Point Five

The Myhill-Nerode theorem is a powerful tool for finding nonregular languages, but it can take some adjusting to get used to.

Let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and consider the following language:

$$L = \{ w \in \Sigma^* \mid \text{there are at least as many } \mathtt{a}\text{'s as } \mathtt{b}\text{'s in } w \}.$$

This language L isn't regular; make sure you have an intuition for why this is.

Below is an attempted proof that L isn't regular. Although the claim it's proving is indeed true, this proof contains an error that renders it incorrect.

> ***Theorem:*** L is not a regular language.
>
> ***Proof:*** Consider the set $S = \{ \mathtt{a}^n \mid n \in \mathbb{N} \}$. This set is infinite because it contains one string for each natural number. We will prove that any two distinct strings in S are distinguishable relative to L. To do so, consider any distinct strings $\mathtt{a}^m$, $\mathtt{a}^n \in S$, and assume without loss of generality that $m > n$. Then $\mathtt{b}^m\mathtt{a}^m \in L$ because this string contains the same number of a's and b's, but $\mathtt{b}^m\mathtt{a}^n \notin L$ because it contains m b's and n a's and $m > n$. Therefore, we see that $\mathtt{a}^m \not\equiv_L \mathtt{a}^n$. This means that S is an infinite set of strings that are pairwise distinguishable relative to L. Therefore, by the Myhill-Nerode theorem, L is not regular. ■

What's wrong with this proof? Be as specific as possible.

The best way to identify a flaw in a proof is to point to a specific claim that's being made that's not true or not properly substantiated and to explain why.

Problem Five: Embracing the Braces

Let Σ be an alphabet containing two characters, the open curly brace character **{** and the close curly brace character **}**. Consider the following language over Σ:

$$L_1 = \{ w \in \Sigma^* \mid w \text{ is a string of balanced curly braces} \}$$

For example, we have **{}** $\in L_1$, **{{}}** $\in L_1$, **{{}{}}{}** $\in L_1$, $\varepsilon \in L_1$, and **{{}}{{{}}}** $\in L_1$, but **}{** $\notin L_1$, **{{}** $\notin L_1$, and **{}}}}** $\notin L_1$. This question explores properties of this language.

i. Prove that L_1 is not a regular language. One consequence of this result – which you don't need

to prove – is that most languages that support some sort of nested structures, such as most programming languages and HTML, aren't regular and so can't be parsed using regular expressions.

As with all problems involving nonregular languages, proceed with this one in stages. First, ask yourself: if you were reading an input string from left to right, what information would you have to keep track of? The Myhill-Nerode theorem asks you to find an infinite set of strings that are all pairwise distinguishable, so try creating an infinite set of strings, one for each possible value that this information could take on.

Let's say that the ***nesting depth*** of a string of balanced braces is the maximum number of unmatched open braces at any point inside the string. For example, the string **{{}{{}}}** has nesting depth three, the string **{{}{}}{}** has nesting depth two, and the string ε has nesting depth zero.

Consider the language $L_2 = \{ w \in \Sigma^* \mid w$ is a string of balanced curly braces with nesting depth at most 4 $\}$. For example, **{}** $\in L_2$, **{{}{}}** $\in L_2$, and **{{{{}}}}{{}}** $\in L_2$, but **{{{{{}}}}}** $\notin L_2$ because although it's a string of balanced curly braces, the nesting goes five levels deep.

ii. Design a DFA for L_2, showing that L_2 is regular. A consequence of *this* result is that while you can't parse all programs or HTML with regular expressions, you can parse programs with low nesting depth or HTML documents without deeply-nested tags using regexes. ***Please submit this DFA using the DFA editor on the course website*** and tell us on GradeScope who submitted it.

iii. Look back at your proof from part (i) of this problem. Imagine that you were to take that exact proof and blindly replace every instance of "L_1" with "L_2." This would give you a (incorrect) proof that L_2 is nonregular (which we know has to be wrong because L_2 is indeed regular.) Where would the error be in that proof? Be as specific as possible.

Again, you should be able to point at a specific spot in the proof that contains a logic error and explain exactly why the statement in question is not true or not supported by the preceding statements. If you can't do this, it likely means you have an error in your proof from part (i)!

Problem Six: State Lower Bounds

The Myhill-Nerode theorem we proved in lecture is actually a special case of a more general theorem about regular languages that can be used to prove lower bounds on the number of states necessary to construct a DFA for a given language.

i. Let L be a language over Σ. Suppose there's a set S, which may be finite and which may be infinite, such that any two distinct strings $x, y \in S$ are distinguishable relative to L (that is, $x \not\equiv_L y$ for any two strings $x, y \in S$ where $x \neq y$.) Prove that any DFA for L must have at least $|S|$ states. (You sometimes hear this referred to as *lower-bounding* the size of any DFA for L.)

According to old-school Twitter rules, all tweets need to be 140 characters or less. Let Σ be the alphabet of characters that can legally appear in a tweet and consider the following language:

$$TWEETS = \{ w \in \Sigma^* \mid |w| \leq 140 \}.$$

This is the language of all legal tweets, assuming the empty string is a legal tweet. The good news is that this language is regular. The bad news is that any DFA for it has to be pretty large.

ii. Using your result from part (i), prove that any DFA for *TWEETS* must have at least 142 states.

It might be easier to tackle this problem if you consider replacing 140 and 142 with some smaller numbers (say, 2 and 4) to build up an intuition. And work backwards – what will you need to do to invoke part (i)?

iii. Define a 142-state DFA for *TWEETS* using the formal 5-tuple definition of a DFA. Briefly explain how your DFA works. No formal proof is necessary.

Again, this might be a lot easier to do if you first reduce 140 and 142 to 2 and 4, respectively, and see what you come up with. Start by drawing out what the DFA would look like, then think about how you'd formalize your idea as a 5-tuple. Remember that to define the transition function, you'll want to use our methods of formally defining a function that we learned earlier in the course; in particular look at how you defined piece-wise functions for Pset4 Q1.

Your results from parts (ii) and (iii) show that the smallest possible DFA for *TWEETS* has exactly 142 states. This approach to finding the smallest object of some type – using some theorem to prove a lower bound ("we need at least this many states") combined with a specific object of the given type ("we can't do worse than this") is a common strategy in algorithm design and computational complexity theory. If you take classes like CS161, CS254, etc., you'll likely see similar sorts of approaches!

Problem Seven: The Extended Transition Function

As you saw on Problem Set Six, formally speaking, a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. You used the 5-tuple definition to pin down edge cases of DFAs. But we can also use this formal definition to rigorously define concepts about automata that, at this point, we've only discussed at a high-level.

Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA. We're going to define a function $\delta^* : \Sigma^* \to Q$ called the ***extended transition function of D***. Intuitively, the function δ^* takes as input a string and outputs what state that string would end up in if run through the DFA D. The function δ^* is defined, recursively, as follows:

- ***Base case:*** $\delta^*(\varepsilon) = q_0$.
- ***Recursive case:*** If $w \in \Sigma^*$ and $a \in \Sigma$, then $\delta^*(wa) = \delta(\delta^*(w), a)$.

That's quite a mouthful, but there's a nice explanation for what's going on here.

i. Explain δ^*'s base case in plain English and why that makes sense given what δ^* represents.

ii. Explain δ^*'s recursive case in plain English and why that makes sense given what δ^* represents.

The notation here is dense, so proceed slowly. What's the input to the function δ^* in this case? What are w and a? You may want to draw out an actual DFA and try expanding out δ^* for a couple of strings.

iii. Explain why, formally speaking, we can define $\mathscr{L}(D) = \{ w \in \Sigma^* \mid \delta^*(w) \in F \}$.

That's a lot of symbols! Write out what each of them mean. Compare this definition against the one from lecture – see if you can account for why this definition has the same meaning.

iv. Let $D = (Q, \Sigma, \delta, q_0, F)$ be a DFA, and let $x, y \in \Sigma^*$ be two strings where $\delta^*(x) = \delta^*(y)$. Prove that, for any string $z \in \Sigma^*$, we have $\delta^*(xz) = \delta^*(yz)$.

Your proof will need to rely on the definition of δ^*. Since δ^* is defined recursively, what style of proof do you think you might want to use here?

Make sure you have an intuition as to what you're asked to prove here. After you peel back the layers of notation, you're left with a nice statement about the behavior of DFAs that's related to something from lecture. While you should use your intuition to guide you, your proof should specifically use the 5-tuple definition of a DFA and the formal definition of the extended transition function. For example, you should not include statements like "run the DFA on xz" or "the DFA ends in an accepting state," since you now have more formal notation available to express those ideas.

In lecture, we used this theorem about distinguishable strings to prove certain languages aren't regular:

> ***Theorem:*** Let x and y be strings where $x \not\equiv_L y$. Then x and y cannot end up in the same state after being run through any DFA for the language L.

We can recast this theorem in terms of the δ* function that we just defined above:

> ***Theorem (Formalized):*** Let x and y be strings where $x \not\equiv_L y$. Then for any DFA D for L, if δ* is the extended transition function for D, we have $\delta^*(x) \neq \delta^*(y)$.

Now that you're equipped with the formal definition of δ*, you can rigorously prove the above statement.

v. Prove the formalized theorem (the second one). Since the goal is to write a rigorous proof of the theorem, you should not cite the informal one from lecture as part of your proof.

Use your intuition about DFAs to think through his one, but as with part (iv) of this problem, use the 5-tuple definition of a DFA and the formal definition of the extended transition function in your proof. For example, you should not use phrases like "run the DFA on x" or "the DFA ends up in an accepting state," since you have more formal notation at your disposal.

Once you've finished, take a minute to marvel at the fact that you're able to read (and prove!) statements like these. Not bad for seven weeks!

Problem Eight: Regular Languages and Equivalence Relations

Throughout this problem set you've been working with the idea that we can take a language L over some alphabet Σ, then work with its distinguishability relation $\not\equiv_L$. A closely related binary relation is the ***indistinguishability*** relation for L, denoted $\equiv_L$. It's also a binary relation over Σ*, and its definition is the negation of the one for distinguishability:

$$x \equiv_L y \quad \text{if} \quad \forall w \in \Sigma^*.\ (xw \in L \leftrightarrow yw \in L).$$

Amazingly, this is always an equivalence relation, regardless of what L is!

i. Prove that if L is a language over Σ, then $\equiv_L$ is an equivalence relation over Σ*.

Whenever you see an equivalence relation, you should immediately start thinking about what its equivalence classes are. Doing so will usually tell you something interesting.

Let's make this more concrete. Let Σ = {a, b} and consider the language M = { $w \in \Sigma^*$ | the number of b's in w is congruent to 1 modulo 5 or to 3 modulo 5 }. For example, a**b**a ∈ M, **b**aaa**b**aaa**b** ∈ M, and **bbbbbb** ∈ M, but aa ∉ M and a**bb**a ∉ M.

ii. How many equivalence classes does the $\equiv_M$ relation have? Briefly describe what those equivalence classes are and give a system of representatives for $\equiv_M$.

Need a refresher on systems of representatives? Check out Problem Set Three.

You might have noticed that each equivalence class of $\equiv_M$ either consists of a bunch of strings not in M or of a bunch of strings that are in M. That's not a coincidence!

iii. Let L be a language over some alphabet Σ and let $x \in \Sigma^*$ be some string. Prove that either *every string* in $[x]_{\equiv_L}$ is in L or that *no strings* in $[x]_{\equiv_L}$ are.

The number of equivalence classes of an equivalence relation is called its ***index***; the index of an equivalence relation R is denoted ***I*(*R*)**. This quantity might be finite, or it might be an infinite cardinality like $\aleph_0$, or even one of the infinities bigger than that.

Armed with the idea of an index, we can state a powerful theorem about finite automata:

> ***Theorem:*** If L is a language over Σ, then every DFA for L must have at least $I(\equiv_L)$ states.

In other words, there's a connection between the number of equivalence classes of a particular binary relation and the minimum sizes of DFAs for that language!

iv. Prove the above theorem. Feel free to use the ***axiom of choice***, which says that every equivalence relation has at least one system of representatives.

Proving this theorem is mostly an exercise in connecting together ideas you've seen used in other places. Think about the relationship between indices and systems of representatives, between distinguishability and indistinguishability, and between what you're doing here and what you've done earlier on this problem set.

There's a very nice intuition for what this theorem says. You can think of the indistinguishability relation for a language L as pinning down the idea "a DFA for L can't tell the difference between these two strings." If you think back to our intuition behind DFA design – build a DFA where each state keeps track of some different piece of information – then you can think of $I(\equiv_L)$ as capturing the number of different pieces of information you'd need to remember. The theorem then says that if you want to build a DFA for a language L, you'll need at least one state per piece of information.

Optional Fun Problem: Generalized Fooling Sets (Extra Credit)

In Problem Seven, you used distinguishability to lower-bound the size of DFAs for a particular language. Unfortunately, distinguishability is not a powerful enough technique to lower-bound the sizes of NFAs. In fact, it's in general quite hard to bound NFA sizes; there's a $1,000,000 prize for anyone who finds a efficient algorithm (for some precise definition of "efficient") that, given an arbitrary NFA, converts it to the smallest possible equivalent NFA!

Although it's generally difficult to lower-bound the sizes of NFAs, there are some techniques we can use to find lower bounds on the sizes of NFAs. Let L be a language over Σ. A ***generalized fooling set*** for L is a set $\mathscr{F} \subseteq \Sigma^* \times \Sigma^*$ is a set with the following properties:

- For any $(x, y) \in \mathscr{F}$, we have $xy \in L$.
- For any distinct pairs (x_1, y_1), $(x_2, y_2) \in \mathscr{F}$, we have $x_1y_2 \notin L$ or $x_2y_1 \notin L$ (this is an inclusive OR.)

Prove that if L is a language and there is a generalized fooling set $\mathscr{F}$ for L that contains n pairs of strings, then any NFA for L must have at least n states.

Don't let the notation scare you off. This is a really cool problem to work through!